The Unix and Internet Fundamentals HOWTO

by Eric S. Raymond
v1.1, 3 December 1998

옮긴이: 권 태 준(linusben@bawi.org)
옮긴날: 13 February 1999

이 문서는 개인용 컴퓨터와 Unix 계열의 운영 체계, 인터넷의 동작 원리에 대해서 기술적으로 쉬운 문체로 서술하고 있다.

목 차

제 1 절 소개

1.1 이 문서의 목적

이 문서는 리눅스와 인터넷을 직접 실행해 가면서 배우고 있는 사용자들에게 도움을 주고자 만들어졌다. 직접 실행해 가면서 배우는 것은 어떤 기술을 습득하는 데 가장 좋은 방법 가운데 하나이지만, 때로는 기초적인 지식에 특별한 허점을 만들기도 한다. 정확한 동작 매커니즘을 알지 목하는 데에서 발생하는 허점(혹은 약점)은 창의적인 생각이나 효과적인 문제 해결을 매우 힘들게 할 수 있다.

나는 이 문서에서 리눅스와 인터넷의 모든 것이 어떻게 작동하는지에 대해서 명확하고 간단하게 말하고자 한다. 이 문서의 모든 예들은 개인용 컴퓨터에서 유닉스와 리눅스를 사용하는 사람들에게 맞게 이루어져 있다. 하지만 이 곳에서 이야기하는 내용들 대부분이 서로 다른 플랫폼이나 다른 유닉스에서도 공통적으로 적용되므로, 앞으로는 시스템을 간단히 '유닉스'라고만 하겠다.

우선 난 여러분이 인텔 개인용 컴퓨터를 사용한다고 가정한다. 만약 여러분이 알파 머신이나 파워PC, 혹은 다른 종류의 유닉스 박스를 사용한다면 세부적인 면에서 약간의 차이가 있을 것이다. 하지만 기본 개념은 모두 같다는 것을 명심하라.

난 어떠한 사항도 반복해서 설명하지 않을 것이기 때문에 여러분은 이 문서를 주의깊게 대해야 할 것이다. 이 말은 당신이 앞으로 읽게 될 모든 단어에서 무언가를 배우게 될 것이라는 것을 의미한다. 처음 읽을 때에는 그냥 대충 훑어봐라 ; 물론 당신이 무엇을 배웠는지를 이해한 다음에 여러차례 다시 읽어야 그 모든 의미를 이해하게 될 것이다.

이 문서는 계속 갱신되는 문서이다. 난 사용자의 피드백에 의해여 새로운 장(section)이 추가되었으면 한다. 따라서 정기적으로 이 문서를 읽어주고 그에 대한 리뷰를 해주길 바란다.

1.2 관계있는 자료들

만약 당신이 해커가 되기 위해서 이 문서를 읽고 있다면, 당신은 또한 아래의 문서를 읽어보기 바란다. *How To Become A Hacker FAQ* `<http://www.tuxedo.org/~esr/faqs/hacker-howto.html>` 이 문서에는 또한 여러가지 유용한 자료들이 링크되어 있다.

1.3 이 문서의 새로운 버전들

The Unix and Internet Fundamentals HOWTO의 새로운 버전은 주기적으로 다음 사이트에 올려질 것이다. `comp.os.linux.help` *news.answers* `<news:answers>`. 이 문서의 새로운 버전은 또한 LDP(Linux Documentation Project) 홈페이지를 비롯한 여러 곳의 리눅스 WWW 그리고 FTP 사이트에 올려질 것이다.

최근의 문서를 월드 와이드 웹에서 찾고 싶다면 아래 사이트를 참고하라. `<http://sunsite.unc.edu/LDP/HOWTO/Fundamentals-HOWTO.html>`.

1.4 피드백과 수정 (Feedback and corrections)

만약 당신이 이 문서에 대해서 질문이나 의견이 있다면, 자유롭게 에릭 레이몬드에게 메일을 보내라. esr@thyrsus.com 어떤 종류의 제안이나 비평을 환영하며, 특히 각각의 개념에 대해 보다 자세한 설명을 추가하는 것을 환영한다. 이 문서의 실수를 발견한다면, 내가 다음 버전에서 그 점을 고칠 수 있도록 나에게 알려주기 바란다.

제 2 절 당신이 가진 컴퓨터의 기본 구조

컴퓨터에는 프로세서 칩이 안에 내장되어 있고, 이것이 실제 연산을 수행한다. 또한 컴퓨터에는 내부 메모리가 존재한다.(DOS/Windows 사용자들은 이것을 'RAM'이라 부르고, 유닉스 사용자들은 'core'라는 표현을 쓴다.) 프로세서와 메모리는 컴퓨터의 심장부에 해당하는 *머더보드*에 설치된다.

컴퓨터에는 스크린과 키보드도 있다. 또한 하드 드라이브와 플로피 디스크도 있다. 스크린과 디스크는 머더보드에 직접 연결되고, 머더보드 밖에 있는 장비들은 *콘트롤러 카드*를 통해서 제어된다. (키보드는 너무 간단해서 따로 카드가 필요없다; 키보드의 콘트롤러는 그 본체 안에 만들어져 있다.)

우리는 뒤에서 이 기계들이 어떻게 작동하는지에 대해서 자세히 살펴볼 것이다. 지금은 먼저 이들이 어떻게 같이 작동할 수 있는지에 대한 약간의 기본적인 지식을 살펴보기로 한다.

컴퓨터 보드 내부의 연결은 *버스*에 의해 이루어진다. 물리적으로, 버스는 콘트롤러(비디오 카드나 디스크 콘트롤러, 사운드 카드 등) 를 꽂는 곳이라 할 수 있다. 버스는 프로세서와 화면, 디스크, 그리고 그 이외의 모든 것들 사이에 데이타 이동이 이루어지는 고속도로와 같은 것이다.

모든 것을 실행 가능하게 하는 프로세서는 컴퓨터의 다른 부분들을 직접적으로 볼 수 없다; 프로세서는 버스를 통해서만 그들과 대화할 수 있다. 프로세스가 빠르게 접근할 수 있는 유일한 서브시스템(sybsystem)은 메모리(the core)이다. 어떤 프로그램이 실행되기 위해서는 그 프로그램이 *core 안에* 있어야 한다.

컴퓨터가 프로그램이나 데이타를 디스크에서 읽어올 때, 실제로 일어나는 것은 프로세서가 디스크 콘트롤러에게 디스크를 읽을 것을 요청하는 메시지를 버스에 보내는 것이다. 얼마 후 디스크 콘트롤러는 자신이 읽은 데이터를 core의 특정 위치에 집어넣었다는 것을 버스를 이용하여 알리게 된다. 그 다음 프로세서는 또다시 버스를 통하여 메모리를 살핀다.

키보드와 스크린 역시 프로세서와 통신을 하기 위해 버스를 통하지만, 다른 주변 기기들보다 그 방식이 더 간단하다. 후에 이들에 대해 다시 논의할 것이다. 지금은 우선 컴퓨터를 켰을 때 컴퓨터 안에서 어떤 일이 일어나는지 알아보자.

제 3 절 스위치를 켰을 때 컴퓨터에는 어떤 일이 일어나는가?

실행되는 프로그램이 없는 컴퓨터는 단지 둔한 전자제품 덩어리에 불과하다. 컴퓨터의 스위치가 켜졌을 때 제일 먼저 해야 할 일은 *운영체계*라 불리우는 특별한 프로그램을 실행시키는 것이다. 운영체계가 하는 일은 복잡한 하드웨어의 제어권을 가지고 다른 프로그램들이 동작하는 것을 돕는 것이다.

운영체계를 불러오는 작업을 *부팅*이라 한다. (원래 *bootstrapping*이란 말에서 유래된 것이고 다른 도움 없이 스스로 어떤 상황을 발전시켜 나아가는 것에 대한 어려움을 나타낸다고 한다.) 컴퓨터는 우선 어떻게 부팅이 되어야 하는지를 알아야 하는데, 이것은 바이오스(BIOS: Basic Input/Output System)라는 칩 안에 그 내용이 담아놓았기 때문에 그곳을 참고하면 알 수 있다.

BIOS 칩은 가장 낮은 숫자를 가진 하드 디스크(*부트 디스크*)의 고정된 위치를 찾아서 *Boot loader*라는 특별한 프로그램을 실행시키려고 한다. (리눅스에서는 boot loader를 LILO라 한다). Boot loader가 찾아지면 이 프로그램이 core로 옮겨와 실행되게 되는데, boot loader는 실제 운영 체계를 실행시키는 일을 한다.

Boot loader는 우선 *커널*을 찾고, 이를 core에 위치시킨 후 실행시킨다. 당신이 리눅스로 부팅하면서 LILO 프롬프트를 화면에서 본 후 여러 개의 점이 화면에 찍히는 것을 본적이 있을 텐데, 이것이 커널을 실행시키

는(loading하는) 과정을 나타낸다. (각각의 점은 커널 코드의 서로 다른 *디스크 블록*을 불러오고 있다는 것을 의미한다.)

(왜 BIOS가 직접 커널을 불러오지 못하는지 궁금해 할지도 모르겠다. – 왜 boot loader와 함께 두 단계의 과정이 필요한 것일까? 우선 BIOS는 그렇게 똑똑하지 못하다. 사실 이것은 매우 멍청하기 때문에, 리눅스는 부팅이 끝나면 이것을 전혀 사용할 수 없게 된다. 이것은 아주 작은 디스크를 가진 원시적인 8비트 컴퓨터를 위해 만들어졌기 때문에 커널을 직접 불러오기에 충분할 정도로 디스크 접근(access)이 힘들다. Boot loader가 작동하는 단계에서는 또한 당신이 서로 다른 디스크 공간에 존재하는 여러 가지 다른 운영체계를 선택적으로 실행시킬 수 있게 해주는데, 유닉스 운영 체계가 충분히 만족스럽지 못하여 다른 운영 체계를 사용하고자 할때 -사실 그런일은 거의 일어나지 않는다.- 사용될 수 있다.)

커널이 시작되면 이것은 다른 하드웨어들을 살피면서 프로그램들을 실행시킬 준비를 한다. 이 작업은 일상적인 메모리 공간에서 이루어지지 않고, *I/O* 포트 (명령을 받기 위한 디바이스 콘트롤러 카드의 특별한 버스 주소) 에서 이루어진다. 커널은 임의로 메모리 공간을 구성하는 것이 아니다; 커널은 어디에서 어떤 정보를 찾을 수 있는지, 콘트롤러가 존재할 때 어떻게 반응하는지 등에 대한 많은 사전 지식을 갖고 있다. 이런 프로세스를 *autoprobing*이라 한다.

부팅이 이루어질 때 보이는 메시지는 대부분이 커널이 하드웨어를 I/O 포트를 통해서 자동검색(autoprobing)을 하는 과정을 보여준다. 이 과정에서 어떤 것이 현재 사용 가능 하며 어떤 것이 적용되는지를 알 수 있다. 리눅스는 이 과정에 있어서 매우 좋은 성능을 가지고 있는데, 대부분의 다른 유닉스보다는 물론이고 도스나 윈도에 비해서도 *아주* 뛰어난 성능을 자랑한다. 사실 리눅스를 오래전부터 사용해오던 사람들 상당수는 부팅시 이루어지는 스마트한 검색(이것은 시스템의 인스톨을 보다 쉽게 만들기도 한다)이, 비판적인 생각을 가지고 있는 많은 사용자들에게 자유로운 유닉스 계열의 시스템이

매력적일 수 있었던 이유 가운데 하나라고 생각했다.

하지만, 커널이 모두 탑재되고 실행되었다고 해서 부트 프로세스가 모두 끝난 것은 아니다; 이것은 단지 첫단계일 뿐이다.(그래서 이 단계를 *run level 1*이라 한다).

커널의 다음 단계는 디스크가 정상인지를 확인하는 것이다. 디스크 파일 시스템은 매우 깨지기 쉬운 것이다; 만일 하드웨어적인 문제나 급작스런 정전 등에 의해 디스크가 손상되었다면, 유닉스 시스템은 정상적인 실행 전에 그것을 확인하고 복구하는 단계가 필요하게 된다. 이 부분에 대한 내용은 후에 8.5(파일 시스템은 어떻게 고장나는가) 부분에서 더 이야기할 것이다.

커널이 다음에 하는 일은 몇개의 *데몬*을 실행시키는 것이다. 데몬이란 프린트 스풀러, 메일 감시자(mail listener)나 웹서버와 같은 프로그램을 말하는데, 이들은 백그라운드에 숨어서 무언가 하기를 기다리고 있는다. 이 특별한 프로그램들은 충돌을 일으킬 수 있는 요청들을 조정하는 역할도 한다. 보통 항상 실행되면서 모든 요청들을 알고 있어야 하는 프로그램을 짜는 것이 수많은 복사본의 프로그램(동시에 실행되면서 각각이 하나씩의 요청을 처리하는)이 서로 충돌하지 않는 것을 확인하려고 하는 것보다 쉽기 때문에 데몬 프로그램들이 존재하게 된다. 시스템이 시작할 때 실행되는 데몬의 집합은 변할 수 있지만, 프린트 스풀러(프린터에 대한 수문장과 같은 데몬)같은 것은 대부분의 경우 포함된다.

모든 데몬들이 시작하게 되면 시스템은 *run level 2*에 접어들게 된다. 다음 단계는 사용자를 위한 준비이다. 커널은 `getty`라는 프로그램을 복사하여 콘솔을 지켜본다 (전화로 접속하는 직렬포트를 같이 지켜보기 위해서는 더 많은 복사본이 필요할 것이다). 이것이 `login`프롬프트가 콘솔에 생기게 하면, 이제 시스템은 *run level 3*에 접어들고 사용자를 받아들이고 프로그램을 실행시키기 위한 준비를 한다.

이름과 비밀번호를 넣고 로그인을 할 때, 당신은 스스로를 `getty`와 컴퓨터에 확인시킨다. 그리고는 (당연히) `login`이라는 프로그램을 실행시키는데, 이것은 보안에 관련된 몇가지 기능을 수행하고 명령어 번역기인 *셸*을 가동시킨다. (물론, `getty`와 `login`은 하나의 프로그램으로 존재할 수 있다. 이들 프로그램이 분리된 데에는 역사적인 원인이 있는데, 이곳에서 다룰 필요는 없다고 생각된다.)

다음 장에서는 셸에서 프로그램을 실행시킬 때 어떤 일이 일어나는지에 대해서 이야기할 것이다.

제 4 절 셸에서 프로그램을 실행시킬 때 어떤 일이 일어나는가?

일반적인 셸에서는 로그인 후에 '$' 프롬프트를 보여줄 것이다.(물론 당신이 이것을 다른 것으로 정해주지 않았을 경우에 성립되는 말이다.) 우리는 이곳에서 셸 문법이나 화면에서 볼 수 있는 것들에 대해서 이야기할 것이 아니다. 컴퓨터의 관점에서 화면 뒤에서 일어나는 컴퓨터의 기작에 대해 이야기하고자 한다.

부팅 후 프로그램을 실행시키기 전의 컴퓨터는 무언가를 하기 위해 기다리고 있는 프로세스들의 '동물원'이라고 생각하면 된다. 이 프로세스들이 기다리고 있는 것은 바로 *이벤트*이다. 이벤트는 키보드를 누르거나 마우스를 움직이면서 발생한다. 혹은 당신의 컴퓨터가 네트워크와 연결되어 있다면 네트워크를 통하여 들어오는 데이터 패킷 역시 하나의 이벤트가 될 수 있다.

커널 역시 이런 프로세스 가운데 하나이다. 하지만 커널은 다른 *사용자 프로세스*가 언제 실행될 것인지를 조절하며, 일반적인 경우 컴퓨터의 하드웨어를 직접 다룰 수 있는 유일한 프로세스이기 때문에 특별하다. 사실 사용자 프로세스는 그들이 키보드로부터 입력을 받거나, 화면에 출력할 때, 디스크에 자료를 쓰거나 읽을 때, 혹은 메모리 안에서 어떤 조작을 할 때 모두 커널에 요청을 해야만 한다. 이런 요청을 *시스템 콜*이라 한다.

일반적으로 모든 I/O는 커널을 거치기 때문에, 실행 및 작동 스케줄이 관리 가능하며 서로간에 프로세스가 충돌하는 일이 없도록 조절될 수 있다. 몇몇 특별한 유저 프로세스 는 곧바로 I/O 포트에 접근할 수 있도록 커널을 비켜가는 것이 허용되기도 한다. X서버들(대부분의 유닉스에서 스크린 그래픽에 대한 다른 프로그램의 요청을 처리하는 프로그램들)이 이런 프로세스의 가장 대표적인 예이다. 하지만 아직 우리는 X서버에 대해 다루지 않을 것이다. 우리는 지금 셸 프롬프트와 문자 콘솔에 대해서 살펴보고 있다.

셸은 단지 하나의 유저 프로세스일 뿐, 특별한 것이 아니다. 이것은 당신이 키보드를 두드리는 것을 기다리면서 (커널을 통하여) 키보드 I/O 포트를 살펴보고 있다. 커널이 그것을 감지하면 입력된 키값을 화면에 보여주고, 그 다음에 셸에 넘겨주게 된다. 커널이 '엔터'를 감지하면 한 행의 텍스트가 셸로 보내진다. 셸은 이것을 하나의 명령어로 다루려고 할 것이다.

ls 를 치고 엔터를 눌러 유닉스의 디렉토리 리스트를 볼려고 한다고 생각해 보자. 셸은 내부에서 정의된 규칙에 따라 '/bin/ls' 파일에 있는 실행 가능한 명령을 수행하려 한다는 것을 알게 된다. 이런 과정에서 커널에 /bin/ls를 새로운 *자식* 프로세스로 시작해 줄 것을 요청하는 시스템 콜이 만들어지고, 이것은 커널을 통하여 키보드와 화면에 접근을 가능하게 한다. 그런 다음 셸은 ls명령이 끝나기를 기다리면서 잠들게 된다.

/bin/ls가 끝나게 되면 커널에 *종료(exit)* 시스템 콜을 보내서 그 사실을 알려준다. 커널은 다시 셸을 깨우고 다시 실행이 가능한 상태로 만든다. 셸은 또다른 프롬프트를 준비하고 또다른 명령어 라인의 입력을 기다리고 된다.

하지만, 당신이 실행시킨 ls가 작동되는 동안에도 다른 것들이 실행될 수 있다. (만약 아주 긴 디렉토리에서 ls를 실행했다고 가정해 보라.) 예를 들면 다른 버추얼 콘솔로 바꾸고 로그인을 해서 퀘이크와 같은 게임을 즐길 수도 있다. 만약 당신의 머신이 인터넷과 연결되어 있다면 /bin/ls가 수행되는 도중에도 메일을 주고받을 수 있는 것이다.

제 5 절 어떻게 입력 도구와 인터럽트가 작동하는가?

키보드는 매우 간단한 입력장치이다; 간단하다는 말은 이것이 작은 양의 데이터를 매우 천천히(컴퓨터의 기준으로 볼 때) 발생시킨다는 것을 의미한다. 키보드를 눌렀다가 뗄 때, 이벤트가 키보드 케이블을 통해 전달되면 *하드웨어 인터프리터*가 발생된다.

이런 인터럽트를 감시하는 것은 운영체계가 해야할 일이다. 모든 가능한 인터럽트에 대해서 *인터럽트 핸들러*가 있어야 하는데, 이것은 운영체계 중에서 인터럽트가 실행될 수 있을 때까지 그와 관련된 데이타(키보드를 누르고 떼는 것과 같은 값)를 따로 모아두는 부분이라 할 수 있다.

키보드에 대해서 인터럽트 핸들러가 실제 행하는 것은 키값을 코어 밑바닥의 시스템 영역에 전달하는 것이다. 그곳에서는 운영 체계가 제어를 현재 키보드 입력을 읽어드릴 프로그램에 직관적으로 넘기는 것을 가능하게 해준다.

디스크나 네트워크 카드와 같이 좀 더 복잡한 입력 도구 역시 비슷하게 작동한다. 위에서 우리는 디스크 콘트롤러가 디스크의 요청을 수행할 때 버스를 이용한다는 것을 살펴보았다. 실제 일어나는 일은 디스크가 인터럽트를 발생시키고, 디스크 인터럽트 핸들러가 후에 데이터를 요청한 프로그램에서 사용될 데이터를 메모리에 복사하게 된다.

모든 종류의 인터럽트는 *우선순위 래벨(priority level)*을 갖는다. 낮은 우선순위를 갖는 인터럽트(키보드 이벤트 같은 것)는 높은 우선순위의 인터럽트(시스템 시간의 흐름이나 디스크 이벤트 등)을 기다려야 한다. 유닉스는 자연스러운 동작을 위해 보다 빠르게 수행되어야 하는 이벤트에게 높은 우선순위를 주도록 설계되어 있다.

당신의 OS가 부팅될 때 볼 수 있는 메시지에는 *IRQ* 넘버라는 것을 볼 수 있을 것이다. 하드웨어의 잘못된 설정 가운데 하나는 정확하게 왜 그런지 모르는 상태에서 서로 다른 장치가 같은 IRQ를 사용하게 하려고 할 때 발생한다.

이 문제에 대한 답이 여기 있다. IRQ는 "인터럽트 요청(INterrupt Request)"의 약자이다. 운영체계는 시작할 때 어떤 인터럽트 넘버가 어떤 하드웨어에서 쓰일 것인지를 알아야 하는데, 그래야만 적당한 핸들러를 각각의 하드웨어에 대해 준비할 수 있기 때문이다. 만약 두개의 서로 다른 하드웨어가 같은 IRQ를 사용하려 한다면 인터럽트는 때때로 잘못된 핸들러에 의해 처리될 것이다. 이런 상황은 적어도 디바이스를 사용 불가능하게 만들고, 때에 따라서는 운영체계를 혼동시켜 시스템 전체를 못쓰게 할 수도 있다.

제 6 절 어떻게 내 컴퓨터는 여러가지 일을 동시에 처리할 수 있는가?

사실은 여러가지 일을 동시에 처리할 수 없다. 컴퓨터는 한 순간에 하나의 작업 (혹은 *프로세스*)만을 수행할 수 있다. 하지만 컴퓨터는 작업을 매우 빠르게 전환할 수 있고, 이것을 인식할 수 없는 사람에게는 여러가지 작업이 동시에 이루어지는 것처럼 생각하게 만든다. 이것을 *시간분할(timesharing)*이라 부른다.

커널의 임무 가운데 하나는 시간분할을 관리하는 것이다. 커널은 이외의 모든 프로세스의 정보를 담고 있는 *스케쥴러(scheduler)*를 가진다. 60분의 1초마다 타이머는 커널에서 시간 인터럽트(clock interrupt)를 발생시키고, 스케줄러는 현재 작업중인 프로세스를 일시 중지시킨 다음에 다른 프로세스를 조정하게 된다.

60분의 1초라는 시간이 매우 큰 시간처럼 들리지는 않을 것이다. 하지만 오늘날의 마이크로프로세서에게는 이것은 수만개의 명령을 실행시키기에 충분한 시간이다. 따라서 당신이 많은 프로세스를 수행하고 있다고 하더라도 각 프로세스는 자신에게 할당된 시간조각(timeslices) 안에서 성공적으로 수행을 마칠 수 있다.

사실 프로그램은 시간조각(timeslice) 전체를 사용할 수는 없다. 만약 인터럽트가 I/O 장치에서 들어온다면, 커널은 현재 작업을 효과적으로 중지시키고, 인터럽트 핸들러를 실행시킨다. 그 후에 현재 작업으로 되돌아가게 된다. 높은 우선순위를 가지는 인터럽트가 많이 발생하게 되면 정상적인 프로세스를 밀어내게 되는데, 이런 비정상적인 행동을 *thrashing*이라 한다. 이것은 현대 유닉스 시스템에서는 일어나기 매우 힘든 행동이다.

실제로 프로그램의 실행 속도는 그 프로그램이 가질 수 있는 기계적인 시간의 양에 제약을 받는 일은 거의 없다. (3차원 그래픽과 사운드같은 경우는 이 규칙의 몇가지 예외에 해당한다.) 프로그램이 늦어지는 일은 디스크 드라이브나 네트워크 연결에서 데이터를 받기위해 프로그램이 기다리는 일 때문이 훨씬 자주 일어난다.

운영체계가 일상적으로 동시에 여러가지 프로세스가 수행할 수 있게 지원하는 경우를 "멀티테스킹(multitasking)"이라 한다. 유닉스 계열의 운영체계는 근본적으로 멀티테스킹이 가능하게 설계되었으며, 멀티테스킹을 훌륭하게 수행한다. – 윈도나 맥 OS보다 유닉스 계열이 훨씬 효과적으로 멀티테스킹을 수행하는데, 이런 운영체계는 멀티테스킹 기능을 후에 추가하였기 때문에 별로 좋은 성능을 내지 못한다. 효과적이고 믿음직한 멀티테스킹은 리눅스가 네트워킹이나 통신, 웹서비스에서 좋은 성능을 발휘할 수 있는 이유 가운데 상당히 큰 비중을 차지하고 있다.

제 7 절 어떻게 컴퓨터에서 서로 프로세스가 충돌하지 않고 잘 돌아가는가?

커널의 스케쥴러는 프로세스를 적절한 시간 간격으로 나누는 일에 신경을 쓴다. 당신의 운영체계 또한 공간적

으로도 프로세스를 구분해야 하는데, 그래야 프로세스들이 사용하는 메모리를 서로 침범하지 않을 수 있기 때문이다. 운영체계에서 이런 문제를 해결하는 것을 *메모리 관리(memory management)*라 한다.

각각의 프로세스는 core 메모리에 자신만의 공간을 필요로 하는데, 그 공간은 프로그램의 변수들과 실행 결과를 저장하고 코드를 실행시키기 위한 공간으로 사용된다. 당신은 이들 세트를 읽기 전용의 *코드 조각(code segment)* (프로세스의 정보를 가지고 있는)과 쓰기 가능한 *데이터 조각(data segment)* (프로세스의 변수 저장)으로 구성되어 있다고 생각할 수 있다. 데이터 조각은 각각의 프로세스에 대해 진정으로 유일하지만, 두개의 프로세스가 같은 코드를 실행시킨다면 유닉스는 자동적으로 시스템의 효율성을 높이기 위해 그 프로세스들이 하나의 코드 조각을 공유하도록 조절한다.

Core 메모리는 비싸기 때문에 효율성이 매우 중요하다. 때때로 머신의 모든 실행 프로그램들이 사용하기에 메모리가 부족한 경우도 생길 수 있다. (X서버와 같이 큰 프로그램을 실행시키는 경우를 예로 들 수 있다.) 이 문제를 해결하기 위해 유닉스는 *가상 메모리(virtual memory)*라는 방법을 쓴다. 이것은 코어에서 실행되는 프로세스의 모든 코드와 데이터를 갖지 않고, 대신에 상대적으로 작은 *작업 세트(working set)*를 갖는다. 프로세스가 해야 할 남은 일은 프로세스의 상태를 당신의 하드디스크에 있는 *스왑공간(swap space)*에 기록하는 것이다.

프로세스가 실행되면 유닉스는 어떻게 작업 세트가 변할것인지 예측을 시도하고 코어에 필요한 조각만을 위치시킨다. 이 작업을 효율적으로 하는 것은 복잡하고 까다로운 작업이기 때문에, 여기서 설명하지는 않기로 한다 – 간단히 말하면, 코드와 데이터 참조(references)가 클러스터에서 작동하려고 하면 새로 생성된 요구는 이전에 혹시 비슷한 것이 있지 않았는지 검색하게 된다. 따라서 만약 유닉스가 아주 자주(혹은 아주 최근에) 이용한 코드나 데이터를 가지고 있는다면, 당신은 시간을 절약할 수 있을 것이다.

두 문단 전에 "때때로"라는 말은 "거의 언제나"를 의미한다는 사실을 주목하라. – 코어의 크기는 전형적으로 실행되는 프로그램의 크기에 비해 작다. 따라서 스와핑은 자주 발생한다. 요즘에는 메모리가 별로 비싸지 않아서 낮은 최종 머신 (low-end machine) 역시 많은 메모리를 가지고 있다. 오늘날의 단일 유저 머신 (single-user machine)이 64MB 이상의 코어를 가지고 있다면, X나 여러가지 작업들을 스와핑 없이 동시에 처리할 수 있다.

이런 행복한 상황에서도, 운영체계의 한 부분인 *메모리 관리자*는 중요한 역할을 한다. 이것은 프로그램들이 자신에게 할당되어 있는 데이터 조각(segments) 만을 사용하는지 확인한다 – 이것은 어떤 프로그램이 다른 프로그램에서 사용하는 데이터를 못쓰게 하는 것을 막아준다.(실수에 의한 것이나 혹은 악의에 의한 것 모두) 테이블은 프로세스가 더 많은 메모리를 요구하거나 자신이 갖고 있던 메모리를 풀어 줄 때마다 갱신된다. (메모리를 풀어주는 일은 보통 프로그램이 종료될 때 일어난다.)

이 테이블은 *MMU(Memory Management Unit)*으로 불리우는 근본적인 하드웨어의 특별한 부분으로 명령을 보내는 데에도 이용된다. 최신의 프로세서 칩들은 그 안에 MMU를 내장하고 있다. MMU는 메모리 주의에 방어벽을 설치하고 그 범위를 넘어서는 참조 시도를 거절하면서 특별한 인터럽트를 발생시킨다.

만약 당신이 "Segmentation fault"나 "core dumped", 혹은 이와 비슷한 같은 유닉스 메시지를 본 적이 있다면 이제 이것이 무엇을 뜻하는지 정확하게 알 수 있을 것이다 ; 실행중인 프로그램이 자신에게 할당된 범위 밖의 메모리를 접근하려고 하다가 치명적인 인터럽트를 발생시킨 것이다. 이건 프로그램상에 버그가 존재하고 있음을 알려준다; *core dump* 의 경우 후에 에러 사항을 진단할 수 있는 정보를 파일로 남겨주기 때문에 프로그래머가 그것을 분석하여 문제를 해결하는 것을 도와준다.

제 8 절 어떻게 컴퓨터에서 데이터를 디스크에 저장하는가?

당신이 유닉스 체제 아래에서 하드 디스크를 바라볼 때, 디렉토리와 파일이라 이름지어진 구조를 보게 될 것이다. 보통 그 이상의 지식이 필요하지는 않지만, 디스크가 고장났을 경우 데이터를 복구하고자 한다면 그 아래에서 어떤 기작이 일어나는지를 알고 있는 것은 유용하다. 불행하게도 파일 단계에서 그 하위 구조로 디스크의 구성을 설명하는 것은 좋은 방법이 아니기 때문에, 나는 이것을 하드웨어 쪽에서 상위 단계로 올라오면서 설명할 것이다.

8.1 하위 단계의 디스크와 파일 시스템 구조

데이타가 저장되는 디스크의 표면은 다트 게임판과 같이 나누어져 있다. – 원형의 트렉(track)과 파이 조각과 같은 섹터(sector)로 나누어져 있다. 바깥쪽 끝의 트렉은 안쪽의 트렉보다 넓기 때문에 보다 많은 섹터로 나누어지게 된다. 각각의 섹터(혹은 *디스크 블록*)은 같은 크기로 이루어져 있고, 오늘날의 유닉스 시스템에서는 일반적으로 1 바이너리 K (8비트 단어 1024개의 분량)로 이루어진다. 각각의 디스크 블록은 고유한 주소를 갖는데 이를 *디스크 블록 숫자*라 한다.

유닉스는 디스크를 *디스크 파티션*으로 나눈다. 각각의 파티션들은 파일 시스템에서 사용하던지 혹은 스왑 공간에서 사용하던지 간에 다른 파티션에서 따로 사용되는 블록의 연속이다. 가장 낮은 번호를 갖는 파티션은 특별하게 취급되는데, 그것은 *부트 파티션*이라 하며 시스템이 시작되기 위해 커널을 넣을 수 있는 파티션이다.

각각의 파티션은 *스왑 공간*(때로는 7(가상 메모리)라 불리운다) 혹은 파일들이 담겨있는 *파일 시스템* 가운데 하나로 사용된다. 스왑 공간으로 사용되는 파티션은 블록이 선형 연결(linear sequence) 로 취급된다. 반면에 파일 시스템은 파일 이름과 디스크 블록의 연결에 대한 지도를 필요로 한다. 파일들이 커지거나 줄어들거나, 혹은 시간에 따라 변화하면서, 파일의 데이타 블록은 선형으로 유지되지 않고 보통 한 파티션 내에 뿔뿔이 흩어지게 된다(운영체계는 자신이 필요로 할 때 비어있는 블록을 찾아낼 수 있는데, 파일들이 새로운 블록을 필요로 할 때에는 운영체계에서 이런 블록들을 건네받게 된다).

8.2 파일 이름과 디렉토리

각각의 파일 시스템에서 이름과 블록 사이의 관계(매핑;mapping)은 *i-node*라 불리우는 구조체를 통해 다루어진다. 이것들은 각각의 파일시스템의 “바닥”(가장 낮은 숫자를 가지는 블록; 실은 그보다 낮은 블록들이 기본적인 정보와 이름을 붙이기 위해(labeling) 사용되는데 이 문서에서는 다루지 않기로 한다) 근처에 집합적으로 존재한다. 각각의 i-node들은 하나의 파일을 기술하고 있으며, 파일 데이타 블록은 inode 위에 존재하게 된다.

모든 i-node는 자신이 기술하고 있는 파일의 디스크 블록 숫자들의 리스트를 가지고 있다. (사실 이 말은 반만 맞다. 작은 파일들에 대해서만 이 말이 성립되지만, 그 이외의 사실은 이곳에서 다루어질 만큼 중요하지 않다.) 여기서 i-node가 파일의 이름을 가지고 있지 *않다*는 것을 주의하기 바란다.

파일 이름들은 *디렉토리 구조* 안에 존재한다. 디렉토리 구조는 파일 이름들과 i-node 숫자들을 연결시켜 주는 지도 역할을 한다. 이것은 유닉스에서 하나의 파일이 여러개의 이름(true name; 혹은 *하드 링크*)을 가질 수 있는지 설명해 준다; 이것은 단지 여러개의 디렉토리 엔트리에서 같은 inode를 가리키고 있는 것이다.

8.3 마운트 지점

가장 간단한 경우, 당신의 모든 유닉스 파일 시스템은 하나의 디스크 파티션에 존재한다. 하지만, 당신이 이런 경우를 개인적 용도의 작은 유닉스 시스템에서 사용되는 것을 본 적이 있을지 몰라도, 이런 방법은 일상적인 것이 아니다. 보다 전형적인 방법은 여러 개의 디스크 파티션이, 서로 다른 물리적 디스크에 나누어져 존재하는 것이다. 따라서, 한 예로, 시스템에 커널이 존재하는 작은 하나의 파티션이 존재한다면, 그보다 약간 큰 파티션에 운영체계 유틸리티들이 존재하고, 보다 큰 파티션에 사용자들의 홈 디렉토리가 존재하는 것이 일반적이다.

커널은 유닉스 파일 시스템의 루트 파티션에서 시작된다. 먼저 루트는 ‘홈(home)’ 디렉토리를 찾는다. 일반적으로 시스템이 부팅된 직후 당신이 접근할 수 있는 것은 유일하게 *루트 파티션*이고, 이것은 (거의 항상) 부팅이 처음 시작되는 곳이기도 하다. 루트 파티션의 파일 시스템에는 루트 디렉토리가 존재하며, 다른 어떤 것보다 우선하는 노드를 가진다.

시스템의 다른 파티션은 이 루트 파티션에 덧붙여지게 되는데, 그럼으로서 모든, 여러 파티션의 파일 시스템이 접근 가능하게 된다. 부트 프로세스 가운데에는 유닉스 시스템이 루트 이외의 다른 파티션을 접근 가능하게 만드는 과정이 포함되어 있다. 이 과정을 통해서 루트 파티션의 디렉토리로 각각의 파티션이 *마운트*되는 것이다.

한 예로, ‘/usr’라는 유닉스의 디렉토리를 가지고 있다면, 이것은 아마 처음 부팅 과정에서는 필요하지 않지만, 시스템에 설치되어 있는 많은 응용 프로그램들을 담고 있는 파티션이 마운트되는 포인트를 가리킨다.

8.4 어떻게 파일은 찾아지는가?

이제 파일 시스템을 상위 레벨에서부터 아래로 바라볼 수 있게 되었다. 당신이 파일(예를 들면, `/home/esr/WWW/ldp/fundamentals.sgml` 파일)을 열 때 다음과 같은 작업이 수행된다.

커널은 유닉스 파일 시스템의 루트(루트 파티션)에서 시작한다. 다음에 이것은 'home'이라 불리는 디렉토리를 찾는다. 일반적으로 'home'은 일반 사용자들이 쓰는 큰 파티션의 마운트 포인트이고, 따라서 그 곳을 찾아가게 된다. 그 유저 파티션의 최상위 단계의 디렉토리 구조에서, 다음에 'esr'이라는 엔트리를 찾고 inode 숫자를 골라낸다. i-node는 이것이 디렉토리 구조라는 것을 알려줄 것이고, 다음에 'WWW'를 찾게 된다. *그것의* i-node를 받은 다음 적당한 하위 디렉토리 'ldp'를 다시 찾아간다. 이것은 다시 적당한 다른 디렉토리의 inode를 줄 것이다. 그것을 연 다음에는 'fundamentals.sgml' 에 대한 i-node 숫자를 갖게 될 것이다. 그 inode는 디렉토리가 아니기 때문에, 파일에 관련된 디스크 블록의 목록을 가지고 있다.

8.5 어떻게 디스크 작동이 잘못될 수 있는가?

앞에서 파일 시스템이 깨지기 쉬운 것임을 얼핏 언급했었다. 이제 우리는 디렉토리와 i-node 레퍼런스의 임의적인 긴 사슬과 같이 파일이 이루어져 있다는 사실을 알게 되었다. 그런데 당신의 하드 디스크에 오류를 포함한 지점이 생겼다면?

당신이 운이 좋다면 단지 몇 개의 파일 시스템이 파괴되었을 것이다. 만약 당신이 운이 좋지 못하다면, 디렉토리 구조나 i-node 숫자가 파괴되었을 수도 있고, 시스템의 하위구조가 모두 지워져 버릴 수도 있다. 혹은, 더 나쁜 경우, 망가진 구조가 같은 디스크 블록이나 inode를 여러가지 방법으로 가리키고 있을 수도 있다.(한마디로 파일 시스템이 얽히는 것이다.) 이런 손상은 정상적인 파일 조작으로 번져나갈 수 있고, 원래 손상된 지점에서 데이터의 손상은 점점 커지게 될 것이다.

다행스럽게도, 이런 종류의 뜻밖의 사고는 하드 디스크가 보다 신뢰성있게 만들어지면서 매우 드문 것이 되었다. 하지만 아직도, 유닉스 시스템은 파일 시스템이 안전하게 보전되어 있는지를 확인하는 integrity-check를 주기적으로 하고 있으며, 그 결과 모든것이 정상적이라는 것을 확인하고 싶어한다. 현대의 유닉스 시스템은 각각의 파티션에 대해 부팅 과정에서 마운트가 이루어지기 전에 무결성을 확인한다. 재부팅을 할 때에는 보다 철저한 검사가 이루어지기 때문에 약간의 시간이 더 걸린다.

만약 이 모든 것들에서 유닉스가 매우 복잡하고 손상받기 쉬운 것이라고 생각된다면, 부팅될 때의 검사로 보통 이 에러들을 잡아낼 수 있고, 또한 그 에러들이 심각한 문제로 발전하기 *전에* 고쳐질 수 있다는 것에서 또한 안심을 하게 될 것이다. 다른 운영체계는 이런 장치가 되어있지 않기 때문에, 부팅이 빠르게 이루어질 수 있지만, 그런 오류를 수동으로 복구하려고 할 때 (당신이 만약 노턴 유틸리티와 같은 프로그램을 처음에 가지고 있었다고 가정할 때 가능한 일이지만) 심각하게 문제가 복잡해질 수 있다.

제 9 절 컴퓨터 언어는 어떻게 작동하는가?

우리는 이미 4(어떻게 프로그램이 작동하는가)애 대해 이야기했다. 모든 프로그램은 컴퓨터가 이해할 수 있는 *기계어*로 만들어 져야만 실행 가능하게 된다. 하지만 사람이 이런 기계어를 직접 다루는 것은 쉽지 않다 ; 이런 일은 해커들에 있어서도 매우 드문 일이고 고도의 기술을 요구하는 것이다.

최근 직접 커널에서 하드웨어 인터페이스를 지원하는 몇몇 프로그램을 제외한 대부분의 유닉스 코드들은 *고급 언어(high-level language)으로 작성된다. ('고급'이라는 말은 역사적으로 기계어와 아주 작은 차이를 갖는 '저급' 어셈블러언어와 구별하기 위해 사용되기 시작한 것이다.)*

고급 언어에는 몇가지 종류가 존재한다. 이것에 대해 이야기하기 위해서는, 먼저 프로그램의 소스 코드(인간이 만든, 편집 가능한 것)가 기계어로 번역되고, 컴퓨터는 그 기계어를 실행시킴으로서 프로그램이 수행된다는 생각을 가지고 있어야 한다.

9.1 컴파일 언어

가장 보편적으로 이야기되는 컴퓨터 언어가 바로 *컴파일언어*이다. 컴파일 언어는 (논리적인) *컴파일러*라는 특별한 프로그램을 이용하여 기계어로 이루어진 실행 가능한 바이너리 파일을 만든다. 한번 실행 파일이 만들어지면, 다시 소스코드를 볼 필요없이 파일을 실행시킬 수 있다. (대부분의 소프트웨어는 컴파일된 바이너리 파일로 배포되고, 소스코드는 볼 수 없다.)

컴파일 언어는 우수한 성능을 보이며 서로 다른 운영체계에서도 실행이 가능하지만, 또한 그만큼 작성하기가 힘들다.

유닉스는 C언어로 작성되어 있는데, C언어(그리고 그 변종인 C++)는 컴파일 언어 가운데 가장 중요한 것이다. FORTRAN은 공학이나 과학 계통에서 아직 많이 쓰이고 있지만, 오래된 언어이기 때문에 좀 원시적이다. 유닉스 시스템에서는 그 이외의 다른 컴파일 언어는 잘 사용되지 않는데, 유닉스 이외의 운영체계에서는 COBOL이 경제/제정 분야에서 널리 사용되고 있다.

그 이외에도 많은 컴파일 언어가 존재하지만, 대부분은 사멸되었거나 제한된 연구에만 이용되고 있다. 만약 당신이 컴파일 언어를 사용하는 새로운 유닉스 개발자가 되고자 한다면 C나 C++과 친해야 한다.

9.2 인터프리터 언어

*인터프리터 언어*은 소스코드를 읽고 그 내용을 통해서 계산이나 시스템 호출이 가능하도록 해주는 인터프리터 프로그램에 의존한다. 소스 코드는 실행될 때마다 매번 번역되어야 하며, 그 때마다 인터프리터가 필요하게 된다.

인터프리터 언어는 컴파일 언어보다 좀 느리고 운영체계나 하드웨어의 접근이 제약을 받기도 한다. 하지만, 인터프리터 언어는 프로그래밍하기가 훨씬 쉽고, 컴파일 언어보다 코드 상의 에러가 미치는 영향이 작다는 장점도 있다.

셸을 포함하여 bc(1), sed(1), awk(1)와 같은 많은 유닉스 유틸리티들이 효율적인 작은 인터프리터 언어이다. 베이직이나 티클(Tcl)도 일반적으로는 인터프리터 언어에 속한다. 역사적으로 가장 중요한 인터프리터 언어는 LISP(그리고 그 계승자들에 의한 발전)가 될 것이다. 오늘날은 Perl이 대중적으로 서서히, 그러나 폭넓게 성장해가는 인터프리터 언어이다.

9.3 P-코드 언어

1990년부터 컴파일과 번역(interpretation)이 같이 사용되는 잡종 언어가 점점 중요하게 되었다. P-코드 언어는 컴파일 언어와 같이 소스코드를 실행 가능한 간단한 바이너리 파일로 만들어주지만, 바이너리 파일은 기계어로 구성되지는 않는다. 대신에 *거짓코드(pseudocode 혹은 p-코드*)로 구성되게 되는데, 이것은 실제 기계어에 비해 간단하지만 더 강력한 기능을 발휘할 수 있다. 프로그램이 실행될 때에는 p-코드가 번역되게 된다.

P-코드는 컴파일된 바이너리 파일과 거의 같은 속도로 실행 가능하다. (p-코드 인터프리터는 매우 간단하고 작지만, 빠르다) 하지만 p-코드 언어는 그 성능이 유동적이고 인터프리터의 성능에 좌우될 수 있다.

중요한 p-코드 언어에는 Python과 자바가 포함된다.

제 10 절 인터넷은 어떻게 작동하는가?

인터넷이 어떻게 작동하는지 이해를 돕기 위해서 당신이 보통 쓰는 인터넷 기능들이 실행될 때 일어나는 일을 살펴볼 것이다 – 문서의 맨 앞에 있는 LDP(Linux Documentation Project) 홈페이지의 이 문서 홈페이지를 살펴보자. 이 문서가

`http://sunsite.unc.edu/LDP/HOWTO/Fundamentals.html`

에 위치하고 있다고 써있다면, 이것은 호스트 sunsite.unc.edu 아래의 웹 디렉토리 가운데 /LDP/HOWTH/Fundamentals.html 파일로 존재한다는 것을 의미한다.

10.1 이름과 위치(Names and locations)

먼저 당신의 브라우저가 해야 할 일은 보고자 하는 문서가 존재하는 컴퓨터와 네트워크를 통해 연결하는 것이다. 이것을 하기 위해서는 우선 sunsite.unc.edu라는 호스트('호스트'는 '호스트 머신' 또는 '네트워크 호스트'의 약자이다; sunsite.unc.edu는 보통 호스트*네임*이라 부른다)가 네트워크 상의 어느 위치에 존재하는지를 알아야 한다. 이 위치에 대응하는 것은 보통 숫자로 이루어져 있는데, 이것을 *IP 어드레스*라 한다('IP'라는 것에 대해서는 후에 다시 설명할 것이다).

이런 작업을 수행하기 위해서 브라우저는 *네임서버*라는 프로그램에 질문을 하게 된다. 네임 서버이 당신 서버에 존재하는 경우도 있지만, 보통은 네임서버 기능을 수행하는 특별한 머신이 존재하고 그곳에 질문을 하게 된다. 만약 당신이 ISP로부터 인터넷 서비스를 받게 되었을 때 설정 과정 가운데 하나는 ISP의 네트워크에 존재하는 네임서버의 IP 어드레스를 인터넷 소프트웨어에 알려주는 것이 될 것이다.

서로 다른 머신에 있는 네임서버들은 상호간에 통신을 하며, 호스트네임을 풀어내기 위한 정보들을 교환하고 새로운 데이타를 갱신한다. 당신의 네임서버는 sunsite.unc.edu라는 이름을 풀기 위해 네트워크 상에 존재하는 서너 군데 다른 사이트에 질문을 하게 되는데, 이 일련의 과정은 매우 빠르게 (보통 1초도 걸리지 않는다) 진행된다.

네임서버는 당신의 브라우저에게 sunsite의 IP 어드레스가 152.2.22.81이라는 것을 알려주게 될 것이다; 이것을 알면 당신의 머신은 sunsite와 정보를 직접 교환할 수 있게 된다.

10.2 패킷과 라우터

브라우저로 Sunsite의 웹서버에 어떤 명령을 보내고자 한다면 다음과 같이 하면 된다:

```
GET /LDP/HOWTO/Fundamentals.html HTTP/1.0
```

이제 실제 어떻게 이것이 동작하는지 살펴보자. 일단 이 명령은 *패킷*으로 만들어진다. 패킷은 전보와 같이 정보의 묶음이라 생각할 수 있는데, 보통 이것은 중요한 세가지 정보로 포장되어 있다; *발송지 주소 (source address)* (당신 컴퓨터의 주소), *목적지 주소(destination address)* (152.2.22.81), 그리고 이것이 월드 와이드 웹의 요청이라는 것을 나타내는 *서비스 번호(service number)* 혹은 *포트 번호(port number)* (이경우 웹서비스의 요청이므로 80)가 그 세가지 이다.

그러면 당신 머신은 만들어진 패킷을 *라우터*라 불리는 특별한 기계에 도착할 때까지 통신선(ISP와 연결된 모뎀선 혹은 지역 네트워크)을 떠돌아다니게 한다. 라우터는 자신의 메모리에 인터넷의 지도를 가지고 있다 – 항상 모든 지도를 메모리에 가지고 있지는 않지만, 당신의 네트워크 주위에 대한 것과 인터넷에 존재하는 다른 이웃의 라우터를 사용할 수 있는 방법을 알고 있다.

당신의 패킷은 목적지에 도달하기 위해 몇몇의 라우터를 거치게 될 것이다. 라우터는 매우 똑똑해서 다른 라우터가 패킷을 받아서 처리하는 데 얼마나 걸리는지를 관찰한다. 그리고는 가장 빠르게 목적지에 도달할 수 있는 연결을 설정하게 된다. 만약 다른 라우터(혹은 케이블)가 사용할 수 없게 되었을 때에도 이 정보를 이용하여 다른 경로를 찾아 패킷을 전송할 수 있다.

일설에 의하면 인터넷이 핵전쟁에도 견딜 수 있도록 설계되었다고 한다. 이것은 사실이 아니지만, 인터넷의 설계는 미지의 장소에서 정상적으로 동작하지 못하는 하드웨어가 있을 때에도 안정적인 성능을 낼 수 있도록 매우 훌륭하게 설계되었다. 이것은 몇개의 집중적인 스위치(전화망처럼)에 의한 것이 아니라 연결에 대한 정보를 수천개의 라우터에 분산시켜 놓았기 때문에 가능한 일이다. 이런 설계는 어떤 문제가 발생했을 때 그 영향을 국지적으로 만들 수 있고, 그 주위의 네트워크는 안전하게 유지될 수 있게 한다.

한번 당신이 보낸 패킷의 목적지까지 도달하게 되면, 목적지의 머신은 서비스 번호를 보고 패킷을 웹서버에 넘기게 된다. 웹서버는 명령 패킷의 발송지 주소 (IP 어드레스)를 보고 어디로 응답을 보내야 할지를 판단한다.

웹서버가 명령에서 요청한 내용을 돌려줄 때, 그 내용은 여러개의 패킷으로 나누어 보내게 된다. 패킷의 크기는 서비스의 종류와 네트워크의 전송 매체의 종류에 따라 다르게 바뀐다.

10.3 TCP와 IP

어떻게 여러 개의 패킷 전송이 관리되는지를 이해하기 위해서, 당신은 인터넷이 실제 두 개의 프로토콜을 이용하며, 그것은 하나가 다른 하나의 상위에 존재하는 구조라는 것을 알 필요가 있다.

아래쪽 단계인 *IP*(Internet Protocol)은 발송지에서 목적지로 보내는 각각의 패킷을 어떻게 취할 것인지에 대한 정보를 담고 있다 (위에서 말한 IP 어드레스를 왜 그렇게 부르는지 이해가 될 것이다.) 하지만, IP는 믿을만한 것은 아니다; 만약 패킷이 분실되었을 경우 발송지나 목적지의 머신 모두 그 사실을 알 수 없다. 전문적인 네트워크 용어로 IP를 *연결이 없는(connectionless)* 프로토콜이라 한다; 보내는 쪽은 단지 패킷을 받는 쪽을 향해 보내지만 그 승인 여부를 확인할 수는 없다.

IP는 하지만 빠르고 경제적이다. 때때로는 신뢰도가 좀 떨어지더라도 빠르고 경제적이라는 것만으로도 충분하다. 만약 당신이 네트워크를 이용해 Doom이나 Quake와 같은 게임을 한다면, 당신이 쏜 총알은 IP 패킷을 통해 표현이 된다. 만약 그 일부가 사라지더라도 그렇게 심각한 문제는 아닐 것이다.

상위 단계인 *TCP*(Transmission Control Protocol)은 이에 비해서 믿을만하다. 두 머신이 TCP로 연결되어 있을 때(TCP 연결은 IP를 이용하여 이루어진다), 수신하는 쪽은 자신이 받은 패킷들에 대한 확인을 송신한 쪽에 보내준다. 만약 송신한 쪽에서 그 확인을 지정된 시간동안 받지 못하면, 그 패킷을 다시 보내게 된다. 게다가 송신하는 쪽은 각각의 TCP 패킷에 일련 번호를 부여하는데, 이것은 수신하는 곳에서 패킷을 정해진 순서로 다시 구성할 수 있게 한다. (만약 연결되어 있는 중에 네트워크가 불안해지면 패킷의 일련번호가 뒤바뀌어 수신될 수 있다.)

TCP/IP 패킷은 또한 잘못된 연결을 통해 발생할 수 있는 데이터의 손상을 확인하기 위해서 체크섬(checksum)을 가지고 있다. 따라서, TCP/IP와 네임서버를 사용하는 사람의 시점에서 보면, 정보가 두 개의 호스트네임/서비스-번호 사이에서 통신되는 것을 신뢰할 수 있다. 내트워크 프로토콜을 제작하는 사람들은 패킷을 만드는 일과, 그 패킷을 다시 구성하는 일, 에러 확인, checksum을 확인하는 일, 그리고 그 아래 단계로 계속 재전달되는 모든 일을 다 고려할 필요가 없다.

10.4 HTTP, 응용 프로토콜

아까 살펴본 예제로 되돌아가 보자. 웹 브라우저와 서버는 TCP/IP의 최상위에서 정보 교환이 이루어지는 응용 프로토콜을 이용하여 대화하게 된다. 응용 프로토콜은 TCP/IP를 이용하여 일련의 정보를 서로 교환하게 된다. 이런 프로토콜을 *HTTP*(Hyper-Text Transfer Protocol)이라 하고, 위에서 살펴본 GET이란 명령어는 프로토콜에서 사용되는 명령어의 한 예라 할 수 있다.

GET 명령어가 sunsite.unc.edu의 웹서버에 서비스 번호 80과 함께 전달되면, 이것은 80번 포트를 관찰하고 있던 *서버 데몬*에 의해 재빨리 처리된다. 이 데몬은 보통 때에는 단지 포트만을 관찰하고 있다가 어떤 명령이 들어오는 경우에만 그 명령을 수행한다.

만약 인터넷이 하나의 전체적인 규칙을 갖도록 설계되었다면, 모든 부분은 매우 간단하고 인간이 쉽게 접근할 수 있었을 것이다. HTTP와 그 비슷한 프로토콜 (호스트 사이에 메일을 주고받게 해주는 Simple Mail Transfer Protocol, *SMTP*도 그 가운데 하나이다.) 은 carriage-return/ line feed로 끝나는 출력 가능한 간단한 텍스트 명령을 이용하는 것이 일반적이다.

하지만 이것은 매우 비효율적이다; 어떤 환경에서는 융통성이 없이 견고하게 짜여진 바이너리 프로토콜이 보다 빠른 성능을 보일 수 있다. 하지만, 실험적으로 인간이 기술하고 이해하기 쉬운 명령어로 이루여졌다는 데에서 오는 장점이 효율성을 높이기 위해 만들어진 까다롭고 복잡한 명령 체계가 가져다 주는 어떤 장점보다 가치있다는 것이 알려져 있다.

따라서, 서버 데몬이 당신에게 TCP/IP를 통해 돌려주는 것 역시 텍스트이다. 그 응답의 시작은 보통 아래와 같이 이루어져 있을 것이다. (몇몇 헤더는 생략되었다):

```
HTTP/1.1 200 OK
Date: Sat, 10 Oct 1998 18:43:35 GMT
Server: Apache/1.2.6 Red Hat
Last-Modified: Thu, 27 Aug 1998 17:55:15 GMT
Content-Length: 2982
Content-Type: text/html
```

이들 해더 뒤에는 빈줄과 웹페이지의 텍스트가 따라올 것이다(연결이 끊어진 후). 당신의 브라우저는 단지 그 페이지를 화면에 보여주기만 한다. 헤더는 그 문서의 상태를 말해준다. (특별히 Content-Type 헤더는 응답으로 돌아온 자료가 HTML인지 말해준다.)